EPiC Series in Computing

Volume 100, 2024, Pages 1–18

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Translating HOL-Light proofs to Coq*

Frédéric Blanqui

Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, **INRIA**
Laboratoire Méthodes Formelles, 4 avenue des Sciences 91190 Gif-sur-Yvette, France

Abstract

We present a method and a tool, hol2dk, to fully automatically translate proofs from the proof assistant HOL-Light to the proof assistant Coq, by using Dedukti as an intermediate language. Moreover, a number of types, functions and predicates defined in HOL-Light are proved (by hand) to be equal to their counterpart in the Coq standard library. By replacing those types and functions by their Coq counterpart everywhere, we obtain a library of theorems (based on classical logic like HOL-Light) that can directly be used and applied in other Coq developments.

1 Introduction

The development of proof assistants made significant progress in the last decades. More and more advanced mathematical theorems are formalized in those proof assistants, including the correctness of complex programs like compilers. Hence, we have bigger and bigger libraries of mathematical theorems. For instance, HOL-Light has a very big library of mathematical theorems in analysis, which Coq does not have. Conversely, Coq has a very big library on algebra, which HOL-Light does not have. But the formalization of all these theorems has taken years of work. It is therefore not reasonable to redo those formalizations in every prover. Instead, one may wonder whether it is possible to transfer results from one prover to the other.

This problem seems to have first been raised in 1996 by Howe, who wanted to reuse results from HOL90 in NuPRL [15]. After that, several researchers tried to translate proofs between specific provers: from HOL98 to Coq [8], from HOL98 to NuPRL [22], from HOL-Light to Coq [30, 19], from HOL-Light to Isabelle [18], from HOL-Light to HOL4 [20], from HOL-Light to Metamath [5], from HOL4 to Isabelle [17], ... Unfortunately, many of these works are not available or not maintained anymore.

HOL-Light received a lot of attention because it is easy to instrument and has interesting libraries. It was used to formalize Hales's proof of the Kepler conjecture, a project called Flyspeck [11]. It is interesting to note that, at the beginning of this project, different parts of Hales's proof were done in three different provers: Coq, Isabelle and HOL-Light. This motivated many works to automatically translate all these parts to a single prover to make sure that their combination was indeed consistent. But it did not work due to the difficulty of automatically

*This publication is based upon work from the action CA20111 EuroProofNet supported by COST (European Cooperation in Science and Technology). The author thanks also the anonymous reviewers for their remarks.

translating proofs from one system to the other, even in the HOL family, and Hales's proof was finally fully formalized by hand in HOL-Light only.

Later, some languages have been developed to serve as common languages for some families of provers based on similar logics, like the OpenTheory format for provers based on Church's simple type theory like HOL, HOL-Light or Isabelle [16]. In 2007, Cousineau and Dowek showed that the larger family of functional pure type systems can be easily encoded into the $\lambda\Pi$-calculus modulo rewriting [7]. This gave rise to the development of the Dedukti logical framework and of translators from Matita, Coq, PVS or Agda to Dedukti and back [29, 9]. A theory in the $\lambda\Pi$-calculus modulo rewriting has been defined which allows the modular representation of many different logics from first-order logic to higher-order logic or the calculus of constructions, in a simple and uniform way [4].

In the present work, we revisit the problem of translating proofs from HOL-Light to Coq, by using Dedukti and building over the works of other authors, mainly [30], [18] and [2].

HOL-Light is a proof assistant based on classical higher-order logic with choice and prenex polymorphism, and implemented using the LCF approach, that is, theorems are built as values of an abstract data type and no independently checkable proof terms are generated. Coq is a proof assistant based on a more complex logic called the predicative calculus of cumulative inductive constructions, featuring first-class polymorphic and dependent types, and where proofs themselves are terms of the logic.

There are no strong theoretical issues in representing HOL-Light proofs in Coq by making explicit a few axioms that are implicit in HOL-Light [30] (see Section 5.1). The difficulties are mainly practical: 1) to be able to handle the huge proofs generated by HOL-Light, 2) to align the types and function definitions of HOL-Light with those of Coq.

Indeed, HOL-Light actually implements a variant of Andrews' Q0 logic [1] where everything is defined from the equality symbol thanks to a few deduction rules, including α-equivalence, the connectives and their introduction and elimination rules. The instrumentation of the kernel of HOL-Light therefore leads to huge proof trees that are then difficult to handle, translate and re-check. For instance, the base library of HOL-Light, hol.ml, which contains 2834 named theorems (see Section 8), uses 14 millions of elementary proof steps taking 5 Go on disk. The HOL-Light library on analysis Multivariate, which contains 16646 more named theorems, uses 182 millions of proof steps taking 120 Go on disk. Even though Coq may be quite efficient for a proof assistant, there is clearly no hope that it can handle a direct and naïve translation of such big proofs with a reasonable amount of time and memory. We therefore need to find ways to a) reduce the size of proofs and b) handle them in parallel. We did so by reusing and extending ideas of [23] and [18].

The second issue, on the alignment of type and function definitions, is not specific to HOL-Light and Coq. Indeed, a translation of proofs from one system to another, which already has some standard library, is really usable only if the theorems we get by translation are relative to the data structures and concepts of that standard library, so that the users of the target system can use them directly. But aligning type and function definitions may require non trivial proofs when those definitions are very different in the source and the target systems, as is the case of HOL-Light and Coq. Indeed, in HOL-Light, Hilbert's choice operator is heavily used to define new types and functions, while Coq primitively features inductive types and recursive functions. We could however formally prove in Coq the equalities of type and function definitions coming from HOL-Light with those of the Coq standard library, for a number of basic types and functions on those types, like natural numbers and lists.

As a consequence, we are now able to automatically obtain readable and directly usable Coq libraries from some HOL-Light libraries. As an example, we published on the Opam repository

of Coq the coq-hol-light library which can be easily installed and used with a single command. It provides 448 lemmas on logic and the theory of arithmetic.

In contrast with previous works, we do not do this translation from HOL-Light to Coq directly but via Dedukti or Lambdapi[1]. We translate definitions and theorems from HOL-Light to Coq in several steps: α) following [25], we patch HOL-Light so that it records proof steps in some `prf` file, β) following [23] and [18], `prf` files are simplified, γ) `prf` files are translated to Dedukti and Lambdapi using some encoding of higher-order logic [4], δ) Dedukti and Lambdapi files using some encoding of higher-order logic are translated to Coq using Lambdapi[2], ε) we prove the correctness of the alignments used in the previous step. The advantage of using these various steps is that some of them can be shared between various translators. For instance, for translating HOL4 proofs to Coq, it is enough to reimplement step α) only, which is specific to HOL4 (and we started some preliminary work in this direction). And for translating HOL-Light proofs to Lean, it is enough to reimplement step δ), and step ε) if it is not done in Lambdapi.

The previous works on the translation of proofs from HOL-Light to Coq we are aware of are [8], [30] and [19]. [8] presents a prototype translating a subset of HOL-Light proof trees represented in some defined proof format (an ancestor of OpenTheory) to Coq proof scripts, each proof step being translated to a tactic call. In [30], Wiedijk does not provide any implementation but show how one could encode HOL-Light proofs in Coq, by translating each HOL-Light proof step as a Coq definition. However, his conclusion is pessimistic on the practicality of this approach w.r.t. Coq ability to handle such big proofs. This led Keller and Werner to try a different approach in [19], namely, to use a deep embedding of HOL-Light proof trees and apply inside Coq a certified boolean function checking their correctness.

Here, we follow Wiedijk's approach [30] and show that, finally, it is feasible when proofs are simplified and translated in parallel. It would be interesting to compare the performances with [19]. Unfortunately, their code is not maintained and does not work out of the box anymore.

The paper is organized as follows: We first explain how HOL-Light proofs are recorded and simplified. Then, we present the logical framework that we use (the $\lambda\Pi$-calculus modulo rewriting) and how we represent recorded proofs in Dedukti and Lambdapi. We discuss the importance of using sharing to represent terms, and of being able to translate proofs in parallel. We then explain how we translate the obtained Lambdapi files to Coq, and prove the correctness of alignments of some type and function definitions between HOL-Light and Coq. We also provide some experimental data about the performance of the various steps.

2 HOL-Light logics and the recording of its proofs

HOL-Light's logic is a variant of Andrews' Q0 logic [1], which itself is a variant of Church's theory of types [6]. The terms of HOL-Light's logic, which are expressed in the programming language OCaml, are those of the simply typed λ-calculus with prenex polymorphism.

In systems like Agda, Coq or Lean, proofs are explicitly represented in memory by λ-terms with holes which are refined step by step by the user (Agda) or via tactics (Coq, Lean) until there is no hole. In systems like HOL-Light, HOL4 or Isabelle, proofs are not explicitly represented in memory nor on disk. These systems are based on the so-called LCF approach [24]. Proved formulas are represented by values of some abstract data type `thm`, and such values can only be obtained by using a few construction functions corresponding to the basic axioms and deduction rules of the logical system at hand.

[1] Lambdapi is a proof assistant compatible with Dedukti and providing additional features like implicit arguments, automated coercion insertion, proof tactics, etc.

[2] https://lambdapi.readthedocs.io/en/latest/options.html#export

In the case of HOL-Light, a value of type `thm`, defined in the file fusion.ml, only contains the formula itself (hypotheses and conclusion) and nothing else:[3]

```
type thm = Sequent of (term list * term)
```

and the construction functions are:

```
val REFL : term -> thm
val TRANS : thm -> thm -> thm
val MK_COMB : thm * thm -> thm
val ABS : term -> thm -> thm
val BETA : term -> thm
val ASSUME : term -> thm
val EQ_MP : thm -> thm -> thm
val DEDUCT_ANTISYM_RULE : thm -> thm -> thm
val INST_TYPE : (hol_type * hol_type) list -> thm -> thm
val INST : (term * term) list -> thm -> thm
```

They correspond to the following deduction rules [13]:

$$\frac{}{\vdash t = t}\ (\text{REFL}) \qquad \frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x, s) = (\lambda x, t)}\ (\text{ABS}) \qquad \frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ (\text{TRANS})$$

$$\frac{}{\{p\} \vdash p}\ (\text{ASSUME}) \qquad \frac{}{\vdash (\lambda x, t)x = t}\ (\text{BETA}) \qquad \frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ (\text{MK\_COMB})$$

$$\frac{\Gamma \vdash p}{\Gamma\theta \vdash p\theta}\ (\text{INST}) \qquad \frac{\Gamma \vdash p}{\Gamma\tau \vdash p\tau}\ (\text{INST\_TYPE}) \qquad \frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ (\text{EQ\_MP})$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q}\ (\text{DEDUCT\_ANTISYM\_RULE})$$

There is no way to know how a theorem has been proved afterwards. To export HOL-Light proofs to other systems, we first need to record how proofs are built. This requires modifying the code of HOL-Light itself: the definition of the type `thm` and of its construction functions. Previous works aiming at exporting HOL-Light proofs, to translate them to other systems or, more recently, to do machine learning experiments, developed such patches:[4]: OpenTheory was developed between 2004 and 2020 for sharing proofs between HOL-based systems [16]; Proofrecording was developed between 2006 and 2010, for exporting HOL-Light proofs first to Isabelle [23] and then to Coq [19]; HOL Import was developed in 2013 for exporting HOL-Light proofs to Isabelle [18]; HOList was developed in 2019 for developing a prover based on machine learning but its code is not accessible anymore [3]; ProofTrace was developed in 2019 by Stanislas Polu for doing machine learning experiments.

When we started this work, we decided to reuse ProofTrace since this was the simplest tool and the only one that was still working, but we ended up simplifying and improving it. The HOL-Light type `thm` is modified as follows:

```
type thm = Sequent of (term list * term * int)
```

where the additional integer argument is a unique identifier. At the beginning, the theorem identifier is set to 0 and is incremented each time a new theorem is created.

In addition, a type `proof` is added, which refers to these identifiers:

[3] OCaml code is written on a green background .

[4] HOL88 also included some export function [31, 32] but it has been replaced by OpenTheory later.

```
type proof = Proof of (thm * proof_content)
and proof_content = Prefl of term | Ptrans of int * int | ...
```

Now, instead of recording proofs in memory (in a hash-table), we directly write them on disk. Hence, we use almost no additional memory. The overhead due to writing proofs on disk is proportionally important but stays quite reasonable: on our machine[5], OCaml takes 2m9s to check and record the proofs of the HOL-Light base library file `hol.ml` instead of 1m14s without proof recording (+74%). But this extra cost at the beginning is largely compensated by the possibility of easily processing proofs written on disk in parallel afterwards (see Section 4). We will however see how to reduce this overhead in the following paragraphs.

In the end, the command `hol2dk dump file.ml` generates several files: `file.prf` is a dump of all the OCaml values of type `proof` generated by HOL-Light; `file.nbp` is a dump of the number of proofs; `file.sig` is a dump of several OCaml values: the types, constants, axioms and definitions (i.e. definitional axioms) introduced by the user; `file.thm` is a dump of a map between theorem identifiers having a name in HOL-Light sources and their name.

For instance, `hol.prf` has 5.5 Go for 14.3 M (millions) proof steps and 2834 named theorems.

The number of proof steps is quite huge. But this is not so surprising after all because HOL-Light's logic is quite low-level. Indeed, the only symbol occurring in HOL-Light rules is the equality symbol $=$: there is no rule for the logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$.

In HOL-Light, a new symbol f can be introduced and latter used by adding an axiom of the form $f = t$ where t is a term where f does not occur, like conjunction:

```
let AND_DEF = new_basic_definition
 `(/\) = \p q. (\f:bool->bool->bool. f p q) = (\f. f T T)`
```

All logical connectives are defined in this way by adding in bool.ml the axioms:

$$\begin{array}{cc}
\top = ((\lambda p, p) = (\lambda p, p)) & \exists = (\lambda p, \forall q, (\forall x, px \Rightarrow q) \Rightarrow q) \\
\wedge = (\lambda p, \lambda q, (\lambda f, fpq) = (\lambda f, f \top \top)) & \vee = (\lambda p, \lambda q, \forall r, (p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r) \\
\Rightarrow = (\lambda p, \lambda q, (p \wedge q) = p) & \bot = (\forall p, p) \\
\forall = (\lambda p, p = (\lambda x, \top)) & \neg = (\lambda p, p \Rightarrow \bot)
\end{array}$$

Introduction and elimination rules of natural deduction are then defined as tactics, that is, functions trying to build a theorem from previous theorems (I say "trying" because some tactic may fail, and this can be exploited when programming tactics). For instance, the introduction and elimination rules for conjunction are defined as follows:

```
let CONJ : thm -> thm -> thm = ...
let CONJUNCT1 : thm -> thm = ...
let CONJUNCT2 : thm -> thm = ...
```

Instrumenting basic tactics. In order to reduce the number of generated proof steps and ease the translation to other proof systems, we decided to also instrument the tactics corresponding to the introduction and elimination rules of connectives in natural deduction, and the tactics corresponding to α-equivalence and β-reduction defined in equal.ml, that is, to record those proof steps as if they were primitive. Indeed, the HOL-Light deduction rule (BETA) for β-reduction is restricted to β-reduction of the same variable, and there is no deduction rule for α-equivalence: in HOL-Light, variable renamings and β-reduction are explicit.

By instrumenting those tactics, we can reduce the number of proof steps to 8.5 M instead of 14.3 M (-40%). OCaml now takes 1m44s to check and record the proofs of the HOL-Light base library file `hol.ml`, instead of 1m14s without proof recording (+40% instead of +74%),

[5] 32 processors Intel(R) Core(TM) i9-13950HX with 36Mo cache and 64Go RAM.

and the generated proof file now has size 3 Go instead of 5.5 Go (-45%). Interestingly, the instrumentation of β-reduction does not contribute to this decrease significantly (<1%). On the other hand, α-conversion counts for half of the decrease.

Simplifying HOL-Light proofs.

Following [23], we can reduce the number of proof steps further by using some rewrite rules:

SYM(REFL(t))	$\hookrightarrow$	REFL(t)	CONJUNCT1(CONJ(p,_))	$\hookrightarrow$	p
SYM(SYM(p))	$\hookrightarrow$	p	CONJUNCT2(CONJ(_,p))	$\hookrightarrow$	p
TRANS(REFL(t),p)	$\hookrightarrow$	p	MKCOMB(REFL(t),REFL(u))	$\hookrightarrow$	REFL(t(u))
TRANS(p,REFL(t))	$\hookrightarrow$	p	EQMP(REFL(_),p)	$\hookrightarrow$	p

Moreover, if, following [18], we remove the useless proof steps that do not contribute to an actual theorem, because they have been generated by a tactic that failed, or because they were part of a proof that has been simplified, the number of proof steps is reduced further by 60%:

initial number of proofs for `hol.ml`	basic tactics instrumentation	simplification and purge
14.3 M	8.5 M (-40%)	3.4 M (-76%)

3 $\lambda\Pi$-calculus modulo rewriting, Dedukti and Lambdapi

The $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$) is a logical framework in which one can encode many logical systems, from first-order logic to higher-order logic, or even complex type systems like the ones of Agda, Coq or Lean, by using just a few symbols and rewrite rules [4].

The terms of the $\lambda\Pi/\mathcal{R}$ are those of the λ-calculus: global symbols $0, +, \ldots$, variables $x, y, \ldots$, function applications $t(u)$ and abstractions $\lambda x : A, t$ where x is annotated by a type A, where types include not only global symbols $\mathbb{N}, \ldots$ and the simple types $A \to B$ for the type of functions from A to B, but also: type-level applications like $Array(x)$ of arrays of size x, type-level abstractions like $\lambda x : \mathbb{N}, Array(x)$, and the dependent types $\Pi x : A, B$ of functions from A to B where B depends on the value of x like the type $\Pi x : \mathbb{N}, Array(x) \to \Pi y : \mathbb{N}, Array(y) \to Array(x + y)$ of the function concatening an array of size x and an array of size y. Actually, $A \to B$ is just a short-hand for $\Pi x : A, B$ when x does not occur in B.

In constrast with simply-typed λ-calculus where types are defined first, and then terms, in this calculus, types depend on terms. In the simply typed λ-calculus, types ensure that terms are well-formed. In the $\lambda\Pi$-calculus, we also need a new kind of types to ensure that the above type expressions are themselves well-formed, hence the following additional syntactic class K of kinds for typing the syntactic class A of usual types: the constant TYPE and the dependent type $\Pi x : A, K$ where K is a kind, like $\mathbb{N} \to$ TYPE for the type of $Array$.

The $\lambda\Pi$-calculus has been studied in details in [12].

$\lambda\Pi/\mathcal{R}$ extends $\lambda\Pi$ by allowing term and type symbols to be defined by a set $\mathcal{R}$ of rewriting rules $l \hookrightarrow r$, that is, oriented equations. Types are then defined modulo those equations (and β-reduction): the types $Array(1 + 1)$ and $Array(2)$ are identified if $1 + 1 \hookrightarrow 2$.

The decidability of type checking in $\lambda\Pi/\mathcal{R}$ depends then on the termination and confluence of the rewriting rules together with β-reduction, and also on the preservation of typing by rewrite rules, a.k.a. the subject reduction property.

Dedukti is a concrete language for expressing $\lambda\Pi/\mathcal{R}$ signatures. It essentially provides two commands: one for adding a new symbol and one for adding a rewriting rule.

Several checking tools for Dedukti exist: Dkcheck [28], Kontroli [10] and Lambdapi [14].

4 From HOL-Light to Dedukti and Lambdapi

For translating HOL-Light proofs to Dedukti and Lambdapi, we reused the encoding of OpenTheory's logic in Dedukti used to implement a translator from OpenTheory to Dedukti [2]. Indeed, OpenTheory's logic is almost the same as the one of HOL-Light.

In the following, we will present the translation to Lambdapi mainly. We will explain the difference with the translation to Dedukti at the end of this section.

The HOL-Light types and terms are defined in OCaml as follows:

```
type hol_type =
| Tyvar of string
| Tyapp of string * hol_type list
```

```
type term =
| Var of string * hol_type
| Const of string * hol_type
| Comb of term * term
| Abs of term * term
```

Two type constructors are pre-defined: the type of propositions `bool` of arity 0, and the type constructor `fun` of arity 2 for representing the type of functions between two types.

HOL-Light types and terms can easily be represented in the following $\lambda\Pi/\mathcal{R}$ signature [4]: a type constant Set:TYPE to represent HOL-Light types, a term constant f:Set→...→Set with n arrows for each HOL-Light type constructor f of arity n, a type-level function El:Set→TYPE interpreting terms of type Set as $\lambda\Pi/\mathcal{R}$ types, and the rule El (fun a b) ↪ El a → El b, interpreting the arrow type of HOL-Light as the arrow type of $\lambda\Pi/\mathcal{R}$.

Then, every HOL-Light type is translated to a $\lambda\Pi/\mathcal{R}$ term of type Set, and every HOL-Light term of type A is translated to a $\lambda\Pi/\mathcal{R}$ term of type El A', where A' is the translation of A. In particular, a proposition is translated to a $\lambda\Pi/\mathcal{R}$ term of type El bool. But such a term is not a $\lambda\Pi/\mathcal{R}$ type. So, to represent the proofs of a proposition, we need to add a type-level function Prf:El bool→TYPE interpreting propositions as types. This allows to represent a proof of a proposition A as a term of type Prf A' by adding a proof constructor for each deduction rule like the term constant REFL : $\Pi\alpha : \mathrm{Set}, \Pi t : \mathrm{El}(\alpha), \mathrm{Prf}(= tt)$ to represent the axiom rule (REFL).

In Lambdapi syntax, this amounts to write the file theory_hol.lp below[6]:

```
/* Encoding of simple type theory */
constant symbol Set : TYPE;
constant symbol bool : Set;
constant symbol fun : Set → Set → Set;
injective symbol El : Set → TYPE;
rule El(fun $a $b) ↪ El $a → El $b;
injective symbol Prf : El bool → TYPE;

/* HOL-Light primitive constants, axioms and rules */
constant symbol = [A] : El(fun A (fun A bool));
symbol REFL [a] (t : El a) : Prf(= t t);
symbol MK_COMB [a b] [s t : El(fun a b)] [u v : El a] :
  Prf(= s t) → Prf(= u v) → Prf(= (s u) (t v));
symbol EQ_MP [p q] : Prf(= p q) → Prf p → Prf q;
...

/* HOL-Light derived connectives */
constant symbol ⇒ : El (fun bool (fun bool bool));
constant symbol ∀ [A] : El (fun (fun A bool) bool);
```

[6] Lambdapi code is written on a blue background .

```
...

/* Natural deduction rules */
rule Prf(⇒ $p $q) ↪ Prf $p → Prf $q;
rule Prf(∀ $p) ↪ Π x,Prf($p x);
symbol ∧i [p] : Prf p → Π[q],Prf q → Prf(∧ p q);
symbol ∧e1 [p q] : Prf(∧ p q) → Prf p;
symbol ∧e2 [p q] : Prf(∧ p q) → Prf q;
symbol ∃i [a] (p : El a → El bool) t : Prf(p t) → Prf(∃ p);
symbol ∃e [a] [p : El a → El bool] :
  Prf(∃(λ x,p x)) → Π[r],(Π x:El a,Prf(p x) → Prf r) → Prf r;
...
```

Arguments written between square brackets are declared implicit and must not be given since the system will try to infer them. Hence, one can write `(= t t)` instead of `(= a t t)`.

The function `Prf` implements the so-called Curry-de Bruijn-Howard correspondence/isomorphism, which allows to reduce proof checking to type checking.

We do not need proof constructors for the introduction and elimination rules of $\Rightarrow$ and $\forall$ because they are derivable thanks to the rewrite rules on `Prf` which interpret the type of proofs of `p⇒q` as the type of functions from the type of proofs of `p` to the type of proofs of `q` (Brouwer-Heyting-Kolmogorov interpretation), and the type of proofs of $\forall$ `p` as the dependent type of functions mapping any `x` of (implicit) type `a` to the type of proofs of `(p x)`.

For representing proofs, we map every HOL-Light deduction rule to some Lambdapi term. The rules (REFL), (MK_COMB) and (EQ_MP) are mapped to the symbols `REFL`, `MK_COMB` and `EQ_MP` declared above. (TRANS) is derivable from the other rules and added in HOL-Light for efficiency reasons only. So it can be derived in Lambdapi too. (BETA) can be translated to (REFL) since, in $\lambda\Pi/\mathcal{R}$, β-equivalent types are identified. For the same reason, (INST) and (INST_TYPE) can be translated to β-redexes: if p is translated to p' and u is translated to u', then $p\{x \mapsto u\}$ is translated to $(\lambda x, p')u'$ which β-reduces to $p'\{x \mapsto u'\}$. Finally, (ABS) and (DEDUCT_ANTISYM_RULE) are implemented using the axioms of functional and proposition extensionality:

```
symbol fun_ext [a b] [f g : El (fun a b)] :
  (Π x, Prf (= (f x) (g x))) → Prf (= f g);
symbol prop_ext [p q] :
  (Prf p → Prf q) → (Prf q → Prf p) → Prf (= p q);
```

Managing identifiers. Names are an important source of difficulties and an important part of the code is dedicated to managing names and renaming them. Because the class of identifiers of each language may be different, we need to replace identifiers of the source language that are invalid in the target language by valid identifiers that are not already used, not too different to ease tracking them back, and not too long for readibility. HOL-Light symbols that are used very often, like logical connectives, are translated to simple and short names or characters for readibility. For instance, the HOL-Light conjunction symbol `/\` is translated to `and` in Dedukti, and $\wedge$ in Lambdapi, taking advantage of the fact that Lambdapi accepts Unicode symbols. We also need to deal with names that are automatically generated by HOL-Light tactics and may contain unusual characters like `%` which are not valid in Coq identifiers. Anticipating the translation to Coq, we decided to replace those characters by `_`. Finally, for the other invalid identifiers we may get, we use a feature of both Dedukti and Lambdapi which accept as identifiers almost any character string enclosed between `{|` and `|}`.

Another difficulty with names is that, in HOL-Light, types and terms live in distinct worlds while their translation leave in the same world. Hence, in HOL-Light, a type symbol and a

term symbol can have the same name. For instance, `sum` is used both as a type (the disjoint sum type) and as a term (the sum operator). In this case, we rename the type symbol by capitalizing its first letter: the `sum` type is renamed into `Sum`.

Worse: in HOL-Light, a variable is defined not only by its name but also by its type: two variables having the same name but different types are distinct. This is not possible in Dedukti, Lambdapi or Coq. Translating every variable to a new identifier made of the name of the variable and its type would generate very long and unreadable identifiers (HOL-Light uses many variables with higher-order types like `fun a (fun a bool)` or `(fun (fun a bool) bool)`). Instead, we locally rename variables whenever it is necessary.

Extra type variables and extra term variables. While translating HOL-Light proofs to Dedukti, it appeared that some proofs had type variables that do not occur in the statement. This means that the statement is true whatever these types are. Hence, it is safe to replace those types by any closed type like `bool`.

It also appeared that some proofs contain term variables that do not occur in the statement. This is not a problem in HOL-Light since, in higher-order logic, it is assumed that every type is inhabited and, indeed, when users want to introduce a new type for some subset of some already existing type, they have to prove that this subset is indeed non empty.

For representing HOL-Light proofs in Lambdapi, we therefore need to add the axiom:

```
symbol el [A] : El A;
```

Then, when we encounter an extra term variable of type A, we can safely replace it by `el A`.

Type and term abbreviations. As already remarked by previous authors, e.g. [23, 18, 2], to get files of reasonable size when translating HOL-Light proofs, it is necessary that the translation preserves some of the sharing implicitly used by the OCaml interpreter when creating terms. However, explicitly sharing subterms between terms having bound variables is difficult. So we implemented the following non-optimal but not too difficult to implement algorithm:

Because we want to identify α-equivalent types, that is, types that are equal modulo renaming of their variables, we start by defining a canonical form for types that is invariant by renaming of their variables. We then incrementally build a data structure injectively mapping canonical types to type identifiers. When encountering a type A with type variables $\alpha_1, \ldots, \alpha_n$ that is not a variable nor a constant, we compute its canonical form A' and check whether A' has some identifier in the map. If not, we add a new mapping between A' and a new type identifier, say $type_k$ obtained by incrementing some counter k. Hence, in both cases, we get an identifier $type_k$ which will be defined in Lambdapi as a function taking as arguments the type variables $\alpha'_1, \ldots, \alpha'_n$ of A' and returning A'. Hence, we can replace the original type expression A by $type_k(\alpha_1, \ldots, \alpha_n)$.

For HOL-Light terms, we proceed similarly though, in this case, term abbreviations are functions taking not only types as arguments but also terms.

Parallel translation and checking. In order to speed up the translation of HOL-Light proofs to Dedukti and Lambdapi, we can take advantage of the fact that nowadays machine have several processors that can be used in parallel. This is possible because one does need to know anything about proof step k when translating proof step k'. The only thing to make sure is that, in the end, the translation of proof step k will appear before the translation of proof step k' if $k < k'$, and this can be easily done by translating disjoint segments of the proof to different files, and compute the dependencies between those segments, so as to be able to use the Linux command make for translating and checking those files in parallel.

Difference between Dedukti and Lambdapi outputs.

- The syntaxes of Dedukti and Lambdapi are different. The Dedukti language allows to represent $\lambda\Pi/\mathcal{R}$ signatures and check their correctness. It is designed to be written and read by

computer programs mainly, while Lambdapi is a proof assistant providing much more features. It is therefore trying to provide a more user-friendly syntax with Unicode characters and infix operators. Lambdapi can however read and generate Dedukti files.

- In Lambdapi, when declaring a new symbol, it is possible to declare some of its arguments as implicit, so that we do not need to write them later since the system will try to infer them automatically. The Lambdapi output of `hol2dk` uses this feature by declaring as implicit all type variables used in HOL-Light symbols and proofs. Hence, the Lambdapi output looks more like the HOL-Light source and gets more readable, while in the Dedukti output all polymorphic symbols must be explicitly applied to the types of their arguments which clutters terms with many types and make them less readable.
- Another difference, that could be fixed in later releases though, is that the Dedukti output is a single huge file, while the Lambdapi and Coq outputs are split in various more manageable files as explained in the previous paragraph. It is however not a problem for the Dedukti file checkers `dkcheck` and `kontroli` as we are going to see in the following section.

4.1 Performance

- Single-threaded translation of the base HOL-Light library `hol.ml`:

output	time	nb files	size	type abbrevs	term abbrevs
dk	9m39s	1	1.3 GB	348 KB	570 MB (44%)
lp	5m04s	7	1.1 GB	308 KB	590 MB (54%)

- Parallel translation using the `split` command:

command	time	nb files	size	type abbrevs	term abbrevs
make -j32 lp	42s	17040	1.1 GB	23 MB	588 MB (53%)

  Handling each theorem separately does not increase the size of term abbreviations significantly. This is not so surprising because each theorem is dealing with different symbols (e.g. there are few terms in common between the theory of list and the theory of reals). On the other hand, it does so for type abbreviations but this remains negligeable w.r.t. the size of proofs, and could be improved by having a single file for all type abbreviations.
- Checking time of the generated Dedukti file: 4m11s with `dk check`.
- Checking time of the generated Lambdapi files: 51m10s with `make -j16 lpo`.
  Lambdapi is much slower than the other checkers. However, it is not necessary for lambdapi to check those files to translate them to Coq: the translation of Lambdapi files to Coq is purely syntactic and thus very fast.

5 Translation of Lambdapi files to Coq

Lambdapi can translate to Coq any `.lp` file written in some encoding of higher-order logic, if it is given the following additional data (see Lambdapi user manual for more details):

- a file `encoding.lp` describing what are the Lambdapi symbols used to encode the symbols `Set`, `bool`, `fun`, `El`, `Prf`, $\Rightarrow$, $\forall$, `=`, $\vee$, $\wedge$, $\exists$, $\neg$ described in the previous section. Essentially, lambdapi removes the symbols `El` and `Prf` and replaces the Lambdapi constructions and the symbols used in the encoding by their Coq counterparts: `Set` is mapped to `Type`, `bool` to `Prop`, `fun` and $\Rightarrow$ to `->`, etc.;

- a file `renaming.lp` providing a finite map to rename Lambdapi identifiers that are invalid in Coq into valid Coq identifiers;
- a file `erasing.lp` providing a finite map from some Lambdapi identifiers to Coq expressions which is used to remove the definitions or axioms whose name are in the domain of the map, and replace everywhere the specified Lambdapi identifiers by their corresponding Coq expressions in the map;
- a Coq file `coq.v` which is required at the beginning of each generated file and can contain definitions and theorems used in `erasing.lp`.

The file `erasing.lp` is key to automatically align the definitions of HOL-Light with those of Coq defined in the Coq standard library, the proof of which must be given in the file `coq.v`.

5.1 Axioms used in Coq

As HOL-Light is based on classical higher-order logic with Hilbert's ε operator, we need to use the following axioms in Coq, which are defined in the Coq standard library:[7]

```
Axiom classic (P : Prop) : P \/ ~ P.
Axiom constructive_indefinite_description (A : Type) P :
  (exists x, P x) -> {x : A | P x}.
Axiom fun_ext {A B: Type} {f g: A -> B}: (forall x, f x = g x) -> f = g.
Axiom prop_ext {P Q : Prop} : (P -> Q) -> (Q -> P) -> P = Q.
Axiom proof_irrelevance (P:Prop) (p1 p2 : P) : p1 = p2.
```

The constant `classic` is the axiom of excluded middle. The constants `fun_ext` and `prop_ext` correspond to functional and propositional extensionality respectively.

HOL-Light's choice operator `@` can be derived from the constant `constructive_indefinite_description` which says that, from any proof of $\exists x, P(x)$, one can extract a witness a and a proof h of $P(a)$. `{x:A|P x}` is the type of dependent pairs (a, h) such that h is a proof of $P(a)$. Indeed, in Coq, proofs are first-class objects and can be used as terms.

The axiom of `proof_irrelevance` says that any two proofs of the same proposition are equal (this does not hold in general as examplified by the development of homotopy type theory). It is used in `coq.v` to prove the correctness of the alignments of some types. Indeed, in HOL-Light, a new type is defined as some isomorphic image of the "subset" of a previously defined type (starting from an undefined type `ind` assumed to be infinite). Following [30], the set of the elements of a type A satisfying some predicate P is translated as the Σ-type `{x:A|P x}`. We therefore need proof irrelevance for (a, p_1) and (a, p_2) to be equal whenever p_1 and p_2 are two distinct proofs of $P(a)$.

5.2 Managing HOL-Light types in Coq

As already mentioned in Section 4, we also have to translate to Coq the Lambdapi axiom `el` saying that every HOL-Light is inhabited. Adding this axiom to Coq would lead to an inconsistent theory because, in Coq, propositions are types and thus types may be empty. As [30, 19], we therefore translate HOL-Light types to a class of non-empty types:

```
Record Type' := { type :> Type; el : type }.
```

where the `>` in the definition of `Type'` tells Coq that the **type** projection function can be implicitly inserted whenever it gets a term of type `Type'` instead of a term of type `Type` [26].

It is then easy to build a value of type `Type'` for each HOL-Light type:

[7] Coq code is written on a yellow background .

```
Definition arr a (b: Type') := {| type := a -> b; el := fun _ => el b |}.
Definition unit' := {| type := unit; el := tt |}.
Definition prod' (a b: Type') := {|type:= a * b; el:= pair(el a)(el b)|}.
Definition nat' := {| type := nat; el := 0 |}.
```

In addition, in order not to clutter the translated statements with those types which are not those of the Coq standard library, we use Coq's canonical structures mechanism[8] by declaring the above types as canonical [27, 21]:

```
Canonical arr. Canonical unit'. Canonical prod'. Canonical nat'.
```

so that the appropriate structure above can be inferred by Coq whenever a term of type `Type` is provided instead a term of type `Type'`. This avoids the need of translating the HOL-Light abstraction to some symbol `hol_Abs` instead of Coq's abstraction itself like in [30].

5.3 Results

The translation of Lambdapi files to Coq is purely syntactic and thus very efficient. It also preserves the structure of files: each `.lp` file is map to a `.v` file. It can therefore be done in parallel by using the Makefile mechanism described in Section 4.

For instance, the 1.1 Go of Lambdapi files generated from `hol.ml` in Section 4.1 can be translated to 1.1 Go of Coq files in 45s only.

The Coq files that we get is quite readable as shown by the following excerpt:

```
Lemma thm_DIV_DIV :
  forall m: nat, forall n: nat, forall p: nat,
  (Nat.div (Nat.div m n) p) = (Nat.div m (Nat.mul n p)).

Lemma thm_MOD_MOD_EXP_MIN:
  forall x: nat, forall p: nat, forall m: nat, forall n: nat,
  (Nat.modulo (Nat.modulo x (Nat.pow p m)) (Nat.pow p n))
  = (Nat.modulo x (Nat.pow p (Nat.min m n))).
```

We could improve the readability further by replacing symbols having an infix notation in Coq by their notation as we already did it for logical connectives (e.g. `Nat.div m n` by `m/n`).

As a final step, to make sure that the translation does not produce incorrect proofs, we need Coq to check the generated Coq files. This is possible but requires quite some time and memory. For instance, checking the translation of the HOL-Light base library file `hol.ml` takes 31m35s with `make -j16 vo`.

5.4 Alignment of HOL-Light and Coq definitions

As explained in the previous section, when translating Lambdapi files to Coq, it is possible to tell Lambdapi to remove some declaration and replace some Lambdapi identifiers by some Coq expressions. This can be used to remove an axiom if this axiom is provable in Coq, and replace a HOL-Light type or function by the corresponding type or function that is defined in the Coq standard library. However, because in HOL-Light any definition of an object `f` gives raise to an axiom of the form `f = t` that is required whenever one wants to use `f`, replacing `f` by some expression `g` defined in Coq is not enough: one also needs to prove that `g = t'` where `t'` is the translation of `t` in Coq.

[8] The author thanks Enrico Tassi for his help on this topic.

To start with, we formally proved in the Coq file coq.v that the following HOL-Light types and constants are indeed equal to their Coq counterparts: all the logical connectives, the HOL-Light deduction rules presented in Section 2, the introduction and elimination rules for natural deduction that are derived in HOL-Light, the unit type and its constructor and eliminator, the product type constructor and its constructor and eliminators, the HOL-Light infinite type `ind` used to define natural numbers together with its constructors and induction principle, the type of natural numbers with its constructors, and the following functions on natural numbers: predecessor, addition, multiplication, power, max, min, substraction, factorial, quotient and remainder in Euclidian division, and the following predicates on natural numbers: $\leq$, $<$, $\geq$, $>$, odd and even.

As examples, we detail the type of natural numbers and the addition function.

In HOL-Light, the type `num` of natural numbers is carved out of some undefined type `ind` which is assumed to be infinite thanks to the following axiom:

```
let INFINITY_AX = new_axiom `?f:ind->ind. ONE_ONE f /\ ~(ONTO f)`
```

From this axiom, new defined symbols are introduced by using Hilbert's choice operator `@`:

```
IND_SUC = @f:ind->ind.
  ?z. (!x1 x2. (f x1 = f x2) = (x1 = x2)) /\ (!x. ~(f x = z))
IND_0 = @z:ind. (!x1 x2. IND_SUC x1 = IND_SUC x2 <=> x1 = x2)
                /\ (!x. ~(IND_SUC x = z))
```

where `?` stands for $\exists$, `!` for $\forall$, and `~` for $\neg$. This means that `IND_SUC` is *some* injective but non-surjective function `f`, and `IND_0` is *some* element not in the image of `IND_SUC`.

The type `num` of natural numbers is then axiomatized as a type isomorphic to the smallest subset of `ind` containing `IND_0` and stable by `IND_SUC`, represented by a predicate `NUM_REP:ind->bool`:

```
let NUM_REP_RULES,NUM_REP_INDUCT,NUM_REP_CASES =
  new_inductive_definition
   `NUM_REP IND_0 /\ (!i. NUM_REP i ==> NUM_REP (IND_SUC i))`

let num_tydef = new_basic_type_definition
  "num" ("mk_num","dest_num") (CONJUNCT1 NUM_REP_RULES)
```

The above line declares two functions `mk_num:ind->num` and `dest_num:num->ind`, and the axioms:

```
  !a:ind, mk_num (dest_num a) = a
  !r:ind, NUM_REP r = (dest_num (mk_num r)) = r)
```

expressing that `dest_num` is injective and surjective on the subset of `ind` satisfying `NUM_REP`.

The constructors of `num` are then defined as follows:

```
let ZERO_DEF = new_definition `_0 = mk_num IND_0`
let SUC_DEF = new_definition `SUC n = mk_num(IND_SUC(dest_num n))`
```

The situation in Coq is very different since the type `nat` of natural numbers is defined by the following simple definition:

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

To replace the HOL-Light symbol `num` by the Coq expression `nat`, the HOL-Light symbol `_0` by the Coq expression `O`, and the HOL-Light symbol `SUC` by the Coq expression `S`, without breaking proofs, we defined in Coq by hand the functions `mk_num` and `dest_num`, and proved that the above axioms are satisfied:

```
Fixpoint dest_num n :=
  match n with 0 => IND_0 | S p => IND_SUC (dest_num p) end.
[...]
Definition mk_num_pred i n := i = dest_num n.
Definition mk_num i := epsilon (mk_num_pred i).
[...]
Lemma axiom_7: forall (a:nat), (mk_num (dest_num a)) = a.
Proof. [...] Qed.
Lemma axiom_8: forall (r:ind), (NUM_REP r) = ((dest_num (mk_num r)) = r).
Proof. [...] Qed.
```

We could similarly align the definition of addition which is defined in Coq as follows:

```
Fixpoint add (n m : nat) : nat :=
  match n with 0 => m | S p => S (add p m) end.
```

The user definition of addition in HOL-Light looks similar to the one of Coq:

```
let ADD = new_recursive_definition num_RECURSION
 `(!n. 0 + n = n) /\ (!m n. (SUC m) + n = SUC(m + n))`
```

But what HOL-Light actually generates relies on the use of the choice operator `@`:

```
`(@add' : ind -> nat -> nat -> nat. !_2155 : nat,
  (!n : nat, add' _2155 _0 n = n) /\ (!m : nat, !n : nat,
    add' _2155 (SUC m) n = SUC (add' _2155 m n)))
((BIT1 (BIT1 (BIT0 (BIT1 (BIT0 (BIT1 0)))))))`
```

(The extra argument built from `BIT0` and `BIT1` is used to tag recursive definitions.)

We however proved in Coq by hand that the two definitions are indeed equal.

By translating the HOL-Light base library file `hol.ml` up to the file `arith.ml` using these alignments, we fully automatically get in a few minutes a small but directly usable Coq library coq-hol-light of 448 lemmas on these aligned functions and predicates like the lemmas `thm_DIV_DIV` and `thm_MOD_MOD_EXP_MIN` mentioned above.

6 Future work

We took the example of the base library hol.ml to provide some data on the performance of translations. But we can also handle in a few minutes other HOL-Light libraries like Logic and Arithmetic which formalize the metatheory of first-order logic and arithmetic.

On the other hand, the library Multivariate, which contains many results on analysis takes many hours to be translated to Lambdapi and rechecked by Coq. The size of the OCaml dump is 120 Go. After simplification, it consists of about 74 millions of useful proof steps (see end of Section 2) for about 27000 theorems (this is half of the whole HOL-Light library).

We plan to extend the alignment of HOL-Light and Coq to the type of real numbers so that this huge HOL-Light library on analysis can benefit to all Coq users.

It would also be interesting to develop Coq tactics to automate the correctness proofs of the alignments of inductive types and recursive functions.

7 Related works

We already briefly discussed some related works in the introduction. We here give more details w.r.t. the most pertinent and recent ones.

- [30] shows how to encode HOL-Light proofs in Coq but does not provide any implementation. We implemented this approach but there are a few differences:
  - We translate the HOL-Light type `bool` to the Coq type `Prop` of propositions instead of the Coq type `bool` of booleans. As a consequence, we do not need to coerce booleans to propositions, but we need to add the axiom of propositional extensionality.
  - We use the axiom of `constructive_indefinite_description` above instead of the apparently more general choice axiom: `forall A:Set, ~~A -> A`.
  - To improve readability of translated statements, we use canonical structures (introduced in Coq 7.2 in January 2002) [27, 21]. It enables Coq to automatically infer an element of a type when a witness is required. As a consequence, we do not need a constructor for abstraction (a HOL-Light abstraction is translated to a Coq abstraction).
  - [30] explores the problem of transfering theorems on the translation of the HOL-Light type of natural numbers `num` to the inductively defined data type `nat` of the Coq standard library, by showing that these two types are isomorphic and that the isomorphism commutes with addition, thus allowing to transfer by hand simple theorems on the translation of `num` to `nat`. We do not transfer theorems from one data type to the other but directly translate one to the other. This however requires to prove that the types and functions are equal.
- [19] uses a deep embedding of HOL-Light proof trees and applies inside Coq a certified boolean function checking their correctness. It would be interesting to compare the performances with our approach. Unfortunately, the code is not maintained anymore.
- [18] describes a tool to extract proofs from HOL-Light, remove useless proof steps and import the obtained proofs in Isabelle.
  - The extraction is done towards a compact portable text file format while we currently use an OCaml-dependent binary file format. We however started to develop a similar text file format in order to extend our tool to HOL4. Thanks to Stéphane Glondu who recently updated HOL Import to more recent versions of OCaml, HOL-Light and Isabelle, we can compare the performances of HOL Import and hol2dk on proof extraction and simplification. For instance, for the HOL-Light library Logic, HOL Import generates a proof file of 233 Mo in 14m14s while hol2dk generates a proof file of 9.9 Go in 12m17s.
  - Subterms are a priori maximally shared but, to reduce memory consumption and improve efficiency, a cache of fixed size is used and, when full, the oldest term is removed.
  - Since HOL-Light and Isabelle are both based on the same logic and implemented using the LCF approach, the import of terms and proofs can be done at the kernel level without having to rename identifiers. This allows to share subterms with bound variables too. This also makes the alignment of definitions easier.
  - No rewrite rule is applied on proofs to simplify them.
  - The import of HOL-Light proofs in Isabelle is not modular.

References

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory*, volume 27 of *Applied Logic Series*. Springer, 2nd edition, 2002.

[2] A. Assaf and G. Burel. Translating HOL to Dedukti. In *Proceedings of the 4th International Workshop on Proof eXchange for Theorem Proving, Electronic Proceedings in Theoretical Computer Science 186*, 2015.

[3] K. Bansal, S. Loos, M. Rabe, C. Szegedy, and S. Wilcox. HOList: An Environment for Machine Learning of Higher-Order Theorem Proving. In *Proceedings of the 36th International Conference on Machine Learning, Proceedings of Machine Learning Research 97*, 2019.

[4] F. Blanqui, G. Dowek, E. Grienenberger, G. Hondet, and F. Thiré. A modular construction of type theories. *Logical Methods in Computer Science*, 19(1):12:1–12:28, 2023.

[5] M. Carneiro. Conversion of HOL Light proofs into Metamath. *Journal of Formalized Reasoning*, 9(1):187–200, 2016.

[6] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[7] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications, Lecture Notes in Computer Science 4583*, 2007.

[8] E. Denney. A Prototype Proof Translator from HOL to Coq. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 1869*, 2000.

[9] G. Dowek and F. Thiré. Logipedia: a multi-system encyclopedia of formal proofs, 2019. Draft.

[10] M. Färber. Safe, fast, concurrent proof checking for the lambda-pi calculus modulo rewriting. In *Proceedings of the 11th International Conference on Certified Programs and Proofs*, 2022.

[11] T. Hales, M. Adams, G. Bauer, T. Dat Dang, J. Harrison, L. Truong Hoang, C. Kaliszyk, V. Magron, S. McLaughin, T. Thang Nguyen, Q. Truong Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, T. Hoai An Ta, N. Trung Tran, T. Diep Trieu, J. Urban, K. Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5(e2):1–29, 2017.

[12] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[13] J. Harrison. HOL Light: An Overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 5674*, 2009.

[14] G. Hondet and F. Blanqui. The New Rewriting Engine of Dedukti. In *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction, Leibniz International Proceedings in Informatics 167*, 2020.

[15] D. Howe. Importing mathematics from HOL into Nuprl. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 1125*, 1996.

[16] J. Hurd. The OpenTheory standard theory library. In *Proceedings of the 3rd International Symposium on NASA Formal Methods, Lecture Notes in Computer Science 6617*, 2011.

[17] F. Immler, J. Rädle, and M. Wenzel. Virtualization of HOL4 in Isabelle. In *Proceedings of the 10th International Conference on Interactive Theorem Proving, Leibniz International Proceedings in Informatics 141*, 2019.

[18] C. Kaliszyk and A. Krauss. Scalable LCF-Style Proof Translation. In *Proceedings of the 4th International Conference on Interactive Theorem Proving, Lecture Notes in Computer Science 7998*, 2013.

[19] C. Keller and B. Werner. Importing HOL Light into Coq. In *Proceedings of the 1st International Conference on Interactive Theorem Proving, Lecture Notes in Computer Science 6172*, 2010.

[20] R. Kumar. Challenges in Using OpenTheory to Transport Harrison's HOL Model from HOL Light to HOL4. In *Proceedings of the 3rd International Workshop on Proof eXchange for Theorem Proving, EasyChair Proceedings in Computing 14*, 2013.

[21] A. Mahboubi and E. Tassi. Canonical Structures for the working Coq user. In *Proceedings of the 4th International Conference on Interactive Theorem Proving, Lecture Notes in Computer Science 7998*, 2013.

[22] P. Naumov, M.-O. Stehr, and J. Meseguer. The HOL/NuPRL Proof Translator. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 2152*, 2001.

[23] S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning, Lecture Notes in Computer Science 4130*,

2006.

[24] L. C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[25] S. Polu. Prooftrace. https://github.com/jrh13/hol-light/tree/master/ProofTrace, 2019.

[26] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[27] A. Saïbi. *Outils génériques de modélisation et de démonstration pour la formalisation des mathématiques en théorie des types. Application á la théorie des catégories*. PhD thesis, Université Paris 6, France, 1999.

[28] R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015.

[29] F. Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Electronic Proceedings in Theoretical Computer Science 274*, 2018.

[30] F. Wiedijk. Encoding the HOL Light logic in Coq. https://www.cs.ru.nl/~freek/notes/holl2coq.pdf, 2007.

[31] W. Wong. Recording HOL proofs. Technical Report UCAM-CL-TR-306, University of Cambridge, Computer Laboratory, 1993.

[32] W. Wong. Validation of HOL Proofs by Proof Checking. *Formal Methods in System Design*, 14:193–212, 1999.

8 Contents of the HOL-Light base library file `hol.ml`

The HOL-Light base library file `hol.ml` which contains 2834 basic theorems on first-order logic, booleans, natural numbers, lists, real numbers, etc.:

```
(* ------------------------------------------------------------------------- *)
(* The logical core.                                                         *)
(* ------------------------------------------------------------------------- *)

loads "fusion.ml";;

(* ------------------------------------------------------------------------- *)
(* Some extra support stuff needed outside the core.                         *)
(* ------------------------------------------------------------------------- *)

loads "basics.ml";;     (* Additional syntax operations and other utilities  *)
loads "nets.ml";;       (* Term nets for fast matchability-based lookup      *)

(* ------------------------------------------------------------------------- *)
(* The interface.                                                            *)
(* ------------------------------------------------------------------------- *)

loads "printer.ml";;    (* Crude prettyprinter                               *)
loads "preterm.ml";;    (* Preterms and their interconversion with terms     *)
loads "parser.ml";;     (* Lexer and parser                                  *)

(* ------------------------------------------------------------------------- *)
(* Higher level deductive system.                                            *)
(* ------------------------------------------------------------------------- *)
```

```
loads "equal.ml";;      (* Basic equality reasoning and conversionals        *)
loads "bool.ml";;       (* Boolean theory and basic derived rules            *)
loads "drule.ml";;      (* Additional derived rules                          *)
loads "tactics.ml";;    (* Tactics, tacticals and goal stack                 *)
loads "itab.ml";;       (* Toy prover for intuitionistic logic               *)
loads "simp.ml";;       (* Basic rewriting and simplification tools          *)
loads "theorems.ml";;   (* Additional theorems (mainly for quantifiers) etc. *)
loads "ind_defs.ml";;   (* Derived rules for inductive definitions           *)
loads "class.ml";;      (* Classical reasoning: Choice and Extensionality    *)
loads "trivia.ml";;     (* Some very basic theories, e.g. type ":1"          *)
loads "canon.ml";;      (* Tools for putting terms in canonical forms        *)
loads "meson.ml";;      (* First order automation: MESON (model elimination) *)
loads "firstorder.ml";; (* More utilities for first-order shadow terms       *)
loads "metis.ml";;      (* More advanced first-order automation: Metis       *)
loads "thecops.ml";;    (* Connection-based automation: leanCoP and nanoCoP  *)
loads "quot.ml";;       (* Derived rules for defining quotient types         *)
loads "impconv.ml";;    (* More powerful implicational rewriting etc.        *)

(* ------------------------------------------------------------------------- *)
(* Mathematical theories and additional proof tools.                         *)
(* ------------------------------------------------------------------------- *)

loads "pair.ml";;       (* Theory of pairs                                   *)
loads "compute.ml";;    (* General call-by-value reduction tool for terms    *)
loads "nums.ml";;       (* Axiom of Infinity, definition of natural numbers  *)
loads "recursion.ml";;  (* Tools for primitive recursion on inductive types  *)
loads "arith.ml";;      (* Natural number arithmetic                         *)
loads "wf.ml";;         (* Theory of wellfounded relations                   *)
loads "calc_num.ml";;   (* Calculation with natural numbers                  *)
loads "normalizer.ml";; (* Polynomial normalizer for rings and semirings     *)
loads "grobner.ml";;    (* Groebner basis procedure for most semirings       *)
loads "ind_types.ml";;  (* Tools for defining inductive types                *)
loads "lists.ml";;      (* Theory of lists                                   *)
loads "realax.ml";;     (* Definition of real numbers                        *)
loads "calc_int.ml";;   (* Calculation with integer-valued reals             *)
loads "realarith.ml";;  (* Universal linear real decision procedure          *)
loads "real.ml";;       (* Derived properties of reals                       *)
loads "calc_rat.ml";;   (* Calculation with rational-valued reals            *)
loads "int.ml";;        (* Definition of integers                            *)
loads "sets.ml";;       (* Basic set theory                                  *)
loads "iterate.ml";;    (* Iterated operations                               *)
loads "cart.ml";;       (* Finite Cartesian products                         *)
loads "define.ml";;     (* Support for general recursive definitions         *)
```